1991

A mixed-mode fault simulator for VLSI MOS devices

David A. Inglis
Lehigh University

AUTHOR:

Inglis, David A.

TITLE:A Mixed-Mode Fault Simulator for VLSI MOS devices

DATE:January 1992

A Mixed-Mode Fault Simulator for VLSI MOS devices

by

David A. Inglis

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Electrical Engineering

Lehigh University

1991

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

Nov 20, 1995

Date

Thesis advisor

CSEE Department Chairperson

Table of Contents

Table of Contents (cont.)

List of Figures

List of Equations

List of Tables

Abstract

With the progression from large scale integration (LSI) to very large scale integration (VLSI), and eventually ultra large scale integration (ULSI), integrated circuit fault coverage is becoming increasingly important. Naturally as we move from LSI towards ULSI the area consumed by devices is getting ever larger. Larger area devices are known to have a higher probability of containing manufacturing induced defects. Fault analysis attempts to measure how well a given set of tests can detect these manufacturing induced defects. One typical means of performing fault analysis is to compare the response of a known "good" (unfaulty) logical circuit, to some stimulus, with that of one containing a fault. However, it can be extremely costly and time consuming to obtain an accurate fault coverage analysis for a VLSI integrated circuit containing many thousands of gates. The cost and time of fault analysis can be contained with a unique technique presented in this thesis. This technique is a serial fault simulation implementation, which uses both the C programming language and switch-level logic simulations, combined, to yield both a cost and time effective method for determining the fault coverage of an integrated circuit.

Chapter 1: Introduction

As integrated circuit design progresses from LSI to VLSI, and eventually ULSI, the measured fault coverage is becoming increasingly important. Fault coverage gives a measure of how well a certain set of test vectors or stimulus is able to detect manufacturing induced defects. There are many commonly used means, today, for modelling these manufacturing defects. In the stuck-at fault model, one of the more widespread models, any node of an integrated circuit can independently be stuck-at-0 or stuck-at-1. However, it can be extremely time consuming and costly to obtain an accurate fault coverage analysis for very large integrated circuits. A typical way around this problem is to use fault sampling in which a random sample of the total number of faults is used to estimate the fault coverage of a circuit. The estimated coverage is, of course, highly dependent upon the size of the given sample. Even with an adequate sample size, for a large VLSI device, there is still the problem of large memory capacity needed to store the "good" model and the "faulty" model(s). Many of the present problems with fault analysis, such as computer run time, cost, and memory capacity, can be overcome if a large portion of the device being analyzed is defined functionally using the C programming language. This thesis is intended to present such a technique, mixed with the use of switch level logic simulation, to help bring the increasing time and cost of performing fault analysis back to manageable levels.

1.1 The Relative Importance of Fault Coverage

Over the past decade or so there have been rapid and vast increases in the complexity, cost, area reduction, frequency, etc. of devices in the semiconductor industry. Semiconductor chip manufacturers have found that in order to make themselves stand out, in this crowded and fiercely competitive arena, their product must be of the highest quality attainable. For this reason, the testing of LSI and VLSI semiconductor devices has taken on a role of increasing importance. Device testing is one of many steps necessary to provide a top quality product. It is also a major portion of the overall product realization process for VLSI devices, as can be seen in a simplified product realization flow chart depicted in Figure 1.

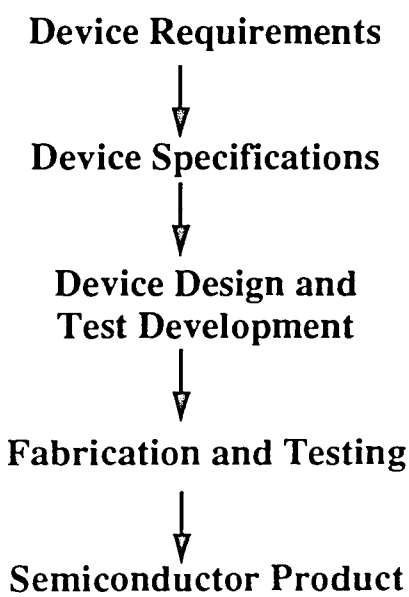


Figure 1. Simplified semiconductor product realization process

It is fairly easy to see that a manufacturer's ability to fully test a particular product is directly related to that products final quality. Semiconductor device testing is very expensive, though, and becoming more so every year.

Containing testing costs is of extreme importance to anyone in the semiconductor business. The cost of testing rises in proportion with the square of the number of individual devices on a chip[1]. This means that as the semiconductor industry moves from one process technology to a future finer line process, which doubles the available device area, the cost and effort to test these devices could quadruple. The advantage of early device testing is evident at the individual chip level but this advantage becomes increasingly important as these chips are used in other products. Reports have been published which put the cost of faulty device detection at a factor of 10 increase for each additional level of assembly[1]. As an illustration of this fact, consider figure 2, which shows the cost of detection of a faulty RAM device at the various levels of assembly.

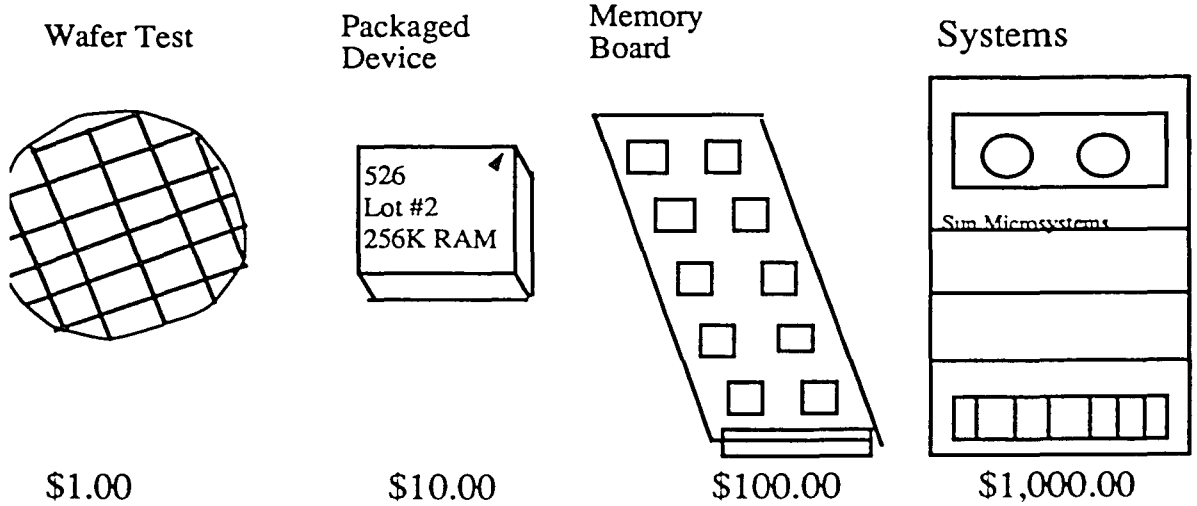


Figure 2. Increasing cost of faulty device detection

The cost of testing a 1K bit RAM chip today, accounts for roughly 10 percent of the total manufacturing cost. The cost of testing a 64K RAM device can be as high as 40 percent of the total cost of manufacturing. Clearly the cost of testing these devices must be curtailed to allow a semiconductor manufacturer to deliver the highest quality part, as well as, make a profit.

1.2 Fault Coverage Quantified

The information, presented in section 1.1, pertaining to chip testing costs, is the prime motivation behind the analysis of manufacturing defects or device faults. Fault coverage is an indication or a measure of how well a set of tests, applied to a device, is able to screen out parts which may contain manufacturing induced defects. This measure serves to evaluate the quality of the tests used on a semiconductor device and is arrived at after performing some method of fault analysis. A high fault coverage gives an increased level of confidence that a particular semiconductor device is being fully tested and that any introduced manufacturing defects will be discovered prior the next level of assembly. Many studies have been done to correlate fault coverage with an overall measure of device quality[2,3,4,5]. This quantitative measure can be arrived at by many different means and is denoted by such terms as reject ratio, yield, PQL (product quality level) and DPM (defects per million). The goal of a very high fault coverage is to yield a low reject ratio for devices, a very high PQL and a very low DPM ratio.

1.2.1 Reject Ratio

A reject ratio is a rather simple notion used to measure a particular product's outgoing

quality level. The reject ratio is defined as the ratio of the number of defective devices, which tested "good", to the total number of devices which tested "good". This relationship can be expressed as

$$\text{Reject ratio} = \frac{\text{Number of faulty devices tested "good"}}{\text{Total number of devices tested "good"}}$$

Figure 3. Reject ratio definition

Semiconductor manufacturers would like to see this ratio as close to zero as possible. A higher percentage fault coverage tends to reduce the numerator of this relationship yielding a lower reject ratio. This simple definition, however, belies the difficulty in applying this measure to semiconductor devices. An actual measure of the reject ratio from tested devices is usually very difficult to obtain and usually not extremely accurate. For this reason, statistical techniques are sometimes used to arrive at a product quality level expressed as a reject ratio. However, the wide variety of models for reject ratio computation raises questions about the validity of the data obtained by these techniques. In a recent study, performed jointly by scientists from AT&T Bell Laboratories, Delco Electronics and the University of Nebraska, five statistical techniques were analyzed[2]. The resultant data indicated reject ratio predictions varied by more than an order of magnitude. Therefore, a

measure of product quality is not always easily and accurately obtained from a predicted reject ratio.

1.2.2 Yield

Another measure of how well faulty devices are screened out is the yield of a device. The yield of a semiconductor wafer is the number of expected good devices for a given area defect density, point defect density and a given active device area. The wafer yield, Y, is expressed by the following equation

$$Y = Y_0 (e)^{-DA} \qquad (1)$$

where Y_0 = area defect yield factor (Large area defects causing whole portions of a wafer to fail)

D = spot defect density (Fatal defects randomly distributed over total wafer)

A = active device area

Two of the main contributors to a reduction in the wafer yield, are the large area defects and the smaller point, or spot defects. Both of these types of process related defects can affect the yield but the spot defects are of greater interest to someone performing fault analysis. Area defects, otherwise known as global defects, tend to affect large areas of the semiconductor wafer and, in many cases, can be easily detected during the manufacturing

process. These area defects are represented by the term Y_0 in equation 1.

Detecting spot defects is typically the reason behind most fault analysis work. Spot defects are randomly located and are caused by local process disturbances. These randomly distributed defects are usually a minute area of either missing or extra material in one of the devices conductive, semi-conductive or insulating layers. Airborne particles are most prevalent among the many sources of spot defects. How these spot defects manifest themselves into various fault models will be discussed in chapter 2. Experimental yield data indicates a best fit if a gamma distribution is assumed for the spot defect density, D. For a very large sample size, the average yield, Y_a, can be expressed by

$$\frac{Y_a}{Y_0} = \left(\frac{1}{(1 + \lambda D_0 A)^{1/\lambda}} \right) \quad . \qquad (2)$$

where Y_0 = average area defect yield factor

D_0 = average value for defect density

λ = variance of the defect density

A = active device area

The effective probed chip area, A_p, as a function of the fault coverage can be expressed

by equation 3. F is the fractional fault coverage obtainable from an applied set of stimuli and A is the active device area.

$$A_p = AF \qquad (3)$$

The true yield, Y_t, of a device is the expected yield given a 100% fault coverage. Given the average yield of equation 2, the ratio of the true yield, Y_t, to the yield at wafer probe, Y_p, can be given by (assuming $Y_t = Y_a$)

$$\frac{Y_t}{Y_p} = (1 + \lambda FAD_0)^{1/\lambda} \bullet (1 + \lambda AD_0)^{\frac{-1}{\lambda}} \qquad (4)$$

Using equation 4, an expression for F can be derived which yields a number for the required fault coverage to achieve an acceptable ratio of true yield to yield at wafer probe. This is expressed in equation 5 below.

$$F = \frac{\left(\left(\frac{Y_t}{Y_p}\right)^{\lambda} \bullet (1 + \lambda D_0 A) - 1\right)}{\lambda D_0 A} \qquad (5)$$

If it is assumed that, λ, the variance of the defect density, approaches zero (indicative of a single spot defect) then equation 2 reduces simply to

$$Y_a = Y_0 (e)^{-D_0 A} \qquad (6)$$

Equation 5 is also reduced to the following simple equation which relates the fault coverage to the expected average wafer yield

$$F = 1 + \frac{1}{D_0 A} \cdot \ln\left(\frac{Y_t}{Y_p}\right) \quad . \qquad (7)$$

Figure 4 shows the fault coverage, F, as a function of Y_t/Y_p for various values of D_0A. Clearly emphasized is the need for very high fault coverage as the semiconductor industry progresses towards VLSI devices which consume ever larger areas, with only minimal decreases in the average defect density.

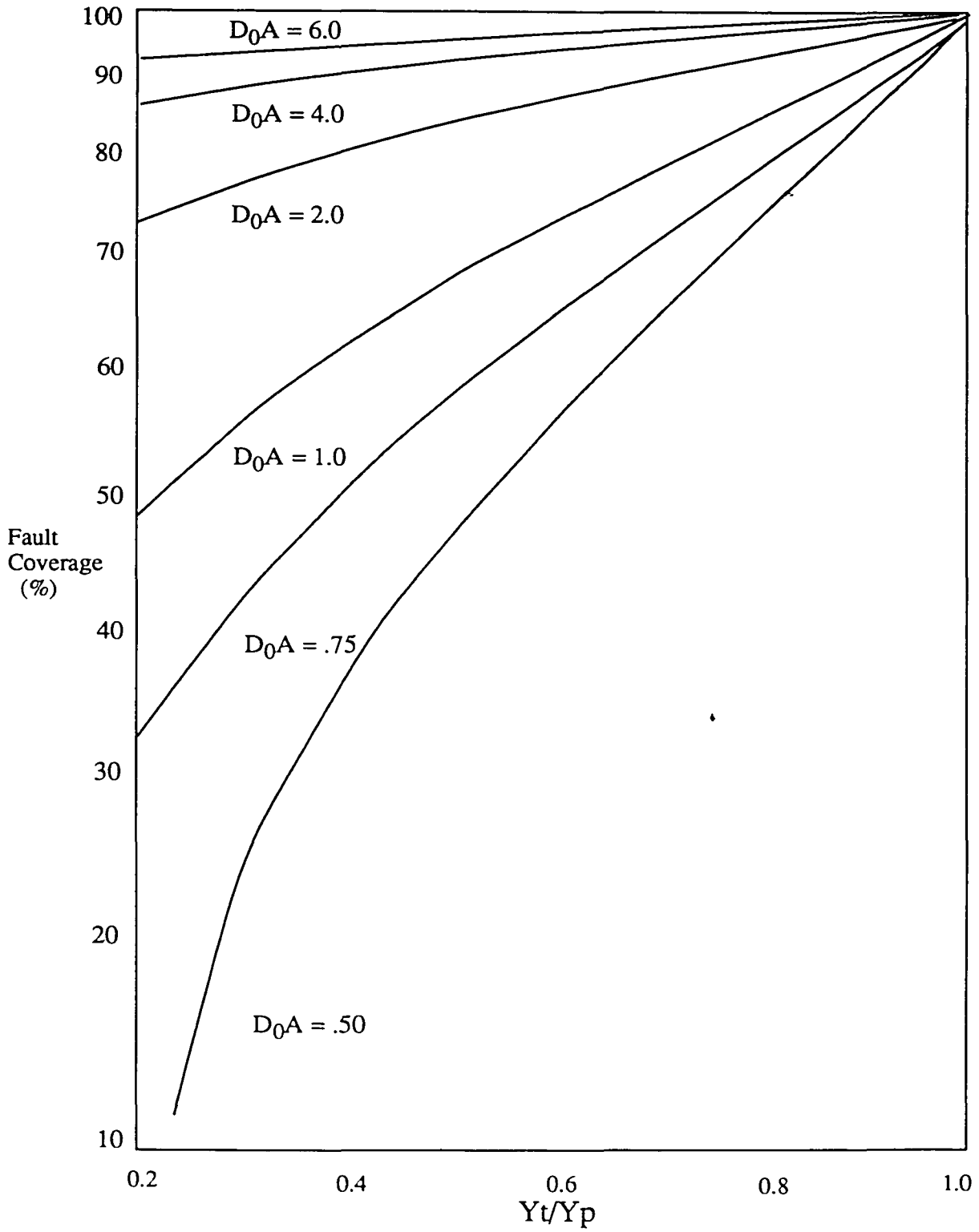


Figure 4. Fault Coverage as a function of Y_t/Y_p and D_0A

1.3 Organization of thesis

The primary goal of this thesis is to present a unique implementation of a mixed-mode fault simulator for VLSI MOS circuits. To fully appreciate the value of such a simulator, some background material needs to be presented on what fault analysis is all about. Chapter 1 gave a brief introduction to the design flow of an MOS device and concrete reasoning for the process of determining the fault coverage of a device. Examples of the cost, to industry, of unacceptably low fault coverage were presented. Chapter 2 introduces many of the fault models that have been presented in the literature and implemented in various fault simulators. These are a few of the many models for manufacturing induced defects, widely known as faults. In Chapter 3, various types of fault simulators are described. The serial, parallel, differential and concurrent type fault simulators are discussed with few advantages and disadvantages for each.

A fault simulation is typically run using a logic simulator with the "good" circuits outputs compared with those of a faulty circuit. Chapter 4 presents three categories of logic simulators. Gate-level, functional-level and switch-level logic simulators are defined with a reasoning behind the development of each type. Basic characteristics of each type of logic simulator are also presented.

Chapter 5 begins the presentation of a unique implementation of a fault simulator using a mixed-mode approach. The two levels of simulation, functional-level and switch-

level, are combined to yield a mixed-mode fault simulator which is fast, accurate and cost effective. The implementation details of this fault simulator are presented in this chapter.

In Chapter 6, the capabilities and shortcomings of the new mixed-mode fault simulator are discussed. Some ideas for future work are also presented.

Chapter 2: Point Defect Models

Point or spot defects are localized manufacturing induced defects. These semiconductor flaws typically are located in very small portions of the overall area but can result in very complex circuit failure modes. There are many sources of these defects, a few of which are airborne particles, particles from the manufacturing equipment, overetches and post-manufacturing sources such as electromigration. These types of flaws can be lone defects and can also occur in clusters within a given device. Because of the numerous failure modes and sources of defects, it is very difficult to completely and accurately define a physical model which covers all point defects. Furthermore, any single model would be both technology and layout dependent. For these reasons there exist many different point defect models being used today for fault analysis[6,7,8]. Some of the more common are the transistor stuck-on, stuck-open, bridges and the stuck-at models.

2.1 Transistor stuck-on fault model

The stuck-on model implies that a transistor is always in the ON or low impedance state. Some possible causes of a stuck-on transistor are gate oxide shorts, bridging defects, punchthrough, parasitic transistor leakage and signal line breaks. This list is by no means exhaustive but gives some of the reasons why a transistor could become permanently conducting. A stuck-on defect can be detected if it causes the logical function of a circuit to differ and this difference is propagated to the external device pins. Many stuck-on defects,

however, are extremely difficult to detect. One clear example of this is a logical NOR gate, shown in Figure 5, with transistor MP1 stuck-on. Even with MP1 stuck-on this logic gate

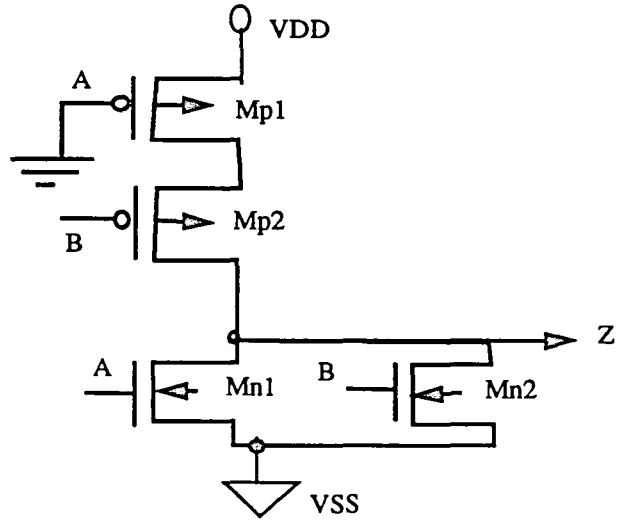


Figure 5. Stuck-on fault at MP1 transistor

would typically behave the same as one without this defect. This is true because PFET's usually have much smaller drive capability than a NFET with similar width to length ratios. In this example the PFET's are also in series.

2.2 Transistor stuck-open fault model

A second physical model used to describe a class of point defects is the stuck-open model. This model is also called the stuck-off or breaks model in the literature. A defect which causes either the gate, drain or source connection of a transistor to become disconnected is covered by this model. Open connections can be caused by missing conducting material or even extra insulating material in the path of the gate, drain or source conductor.

An important point concerning the stuck-open model is that this type of defect may occur both during device fabrication and as a latent failure. Latent failures can arise from electomigration of conducting material and conductor voids due to rapid thermal expansion and contraction. Implementing this model is difficult due to the large number of connecting layers and conductors which can suffer "opens". This model must cover breaks in semiconductor material internal to a logic gate (intra-gate breaks) as well as those external to any logic gate (signal-line breaks). Intra-gate opens have markedly different characteristics than signal-line breaks and yet both must be encompassed by this model.

An intra-gate open, which leaves no low impedance path between the logic gates output and VSS or VDD, will yield a gate who's output is stuck-at either high or low. If, however, all low impedance paths to either VDD or VSS are not broken then the logic gate behaves like a dynamic sequential circuit. Here alone, different faulty behavior is observed for just slightly different intra-gate defects.

The signal-line breaks result in transistors with floating gates. Depending on the charge stored on the transistor gate, this can result in either a permanently non-conducting transistor or a permanently conducting transistor. The non-conducting transistor will behave similar to an intra-gate defect while the permanently conducting transistor will behave as the stuck-on model.

2.3 Delay fault model

A slightly different type of fault model is the delay fault model. This fault model at-

tempts to detect some continuous parameter of the circuit such as path delays. A fault of this type would be considered a parametric fault, whereas the previously discussed models are all considered logical faults. Any predefined increase in the time it takes for a specific set of test stimuli to propagate through a faulty device would be defined as a delay fault. Delay faults can be caused by subtle modifications to a semiconductor process or other types of faults, such as bridging faults.

2.4 Bridging fault model

A bridge fault model is a model of the physical effect of two or more circuit nodes which become bridged or connected to one another due to some fabrication related defect. Like the stuck-open fault, bridges can be caused by either missing insulating material or extra conductive material deposited upon a semiconductor device. This fault model is especially complex to implement as it can only be detected if the bridged nodes are at different logic levels. Two bridged nodes at a logic level "1" will behave the same as two unbridged nodes both held at a logic "0" level. Not only must the circuit nodes be at different potentials, but one node must be able to overdrive the other and then to propagate this faulty behavior to the circuit outputs where the fault can be observed. However, even if one node does not completely overpower the other, this bridge fault may still slow the bridged nodes sufficiently so that this fault may be detected as a delay fault. Another problem with this fault model is that the number of possible bridges in a circuit is often pro-

hibitively large. Considering only bridges between two nodes, there are theoretically on the order of n^2 pairs of nodes which could be bridged, where n is the number of circuit nodes.

2.5 Stuck-at fault model

By far the simplest model, and the one most commonly used in fault simulation, is the single stuck-at model. In this point defect model, any node of a circuit can independently become stuck at logic "1" or logic "0". A stuck-at zero fault models a circuit node permanently connected to ground potential while a stuck-at one fault models a circuit node permanently connected to the circuit power. This is considered the "classical" fault model and it has gained widespread use in fault simulation due to the following reasons:

- stuck-at faults are technology independent;
- they effectively represent most manufacturing induced defects;
- they can be analyzed by known methods; and
- they have proven to be an effective measure of test quality.

The stuck-at fault model strikes a fair balance between a model which is simple enough to implement in a fault simulator and yet still complex enough to represent a majority of the manufacturing defects which may occur. It is technology independent since the model is not concerned with the construction of the gates themselves. This fault model

also allows for a technique called fault collapsing which reduces the number of faults which need to be simulated, by removing equivalent faults from the total stuck-at fault list. For example, the 2 input NAND gate, depicted in Figure 6, could have a total of 6 stuck-at faults. Each input could be stuck-at logic "1" or stuck-at logic "0". In addition, the output, Z, can also be stuck-at logic "1" or logic "0". It can be easily seen that this circuit will have the same faulty behavior if the output is stuck-at logic "0" or if either input, A or B, is stuck-at logic "1". This logic gate will also have the same faulty behavior if either input is stuck-at logic "0" as the output being stuck-at logic "1". This example clearly

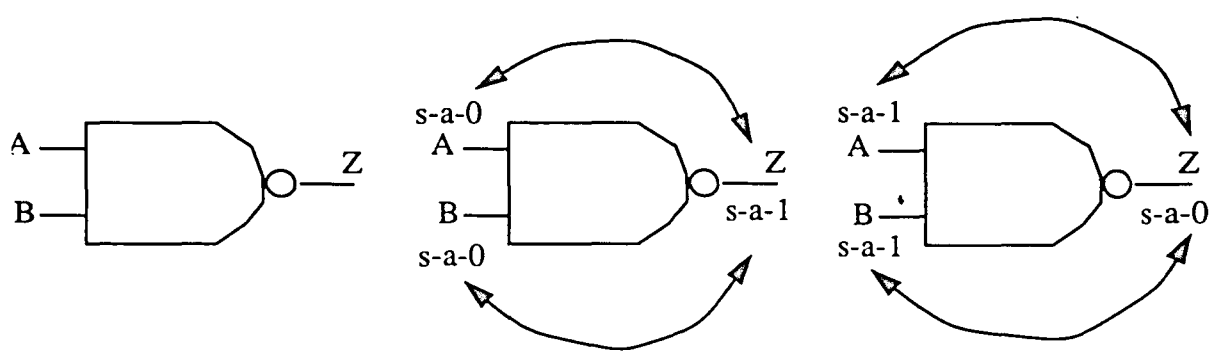


Figure 6. Stuck-at fault for a 2 input NAND gate

indicates the value of fault collapsing in reducing the overall size of the fault list.

It is obvious that the number of stuck-at faults for a relatively small circuit can become quite large. The fault collapsing technique, however, brings the number of faults, which must be simulated, back down to a manageable level. This type of fault model is typically

applied at the gate level and it is implemented in many commercial fault simulators, including AT&T Bell Laboratories fault simulator MOTIS/FSIM[9]. This same front end tool, which performs the fault list generation and fault collapsing, is used in the serial fault simulator presented in this thesis.

The stuck-at fault model is implemented in the fault simulator presented in this thesis, primarily due to its ease of implementation. This fault model only requires the ability to find the faulty node, in the circuit netlist, and tie it to either logic "0" or logic "1". There is, however, a fair amount of data and experience, with this fault model, which indicates that it covers the majority of the manufacturing induced defects.

Chapter 3. Types of fault simulators

The measurement of fault coverage is typically done with the use of a fault simulator. A given fault simulator may implement any of the previously discussed fault models within the chosen fault algorithm. A fault algorithm is chosen by making trade-offs between ease of implementation, performance objectives, memory capacity, CPU power, etc. Several algorithms have been proposed, for VLSI semiconductor design, and implemented in commercial fault simulators. These fault simulators will generally fall into one of the following categories:

- Serial fault simulation
- Parallel fault simulation
- Deductive fault simulation
- Differential fault simulation
- Concurrent fault simulation

Another promising area of work in fault detection, although not simulation based, is the recent work in quiescent current (IDDQ) monitoring or current testing[10,11,12,13]. This technique takes advantage of the fact that CMOS circuits are designed to draw very little quiescent current from the power supplies. IDDQ monitoring is especially useful in sequential circuits where many stuck-at faults may go undetected.

3.1 Serial fault simulation

The serial or sequential fault simulation algorithm is simple and easy to implement. This algorithm models the responses of N + 1 circuits, these being the "good" circuit and N faulty circuits. One logical fault is inserted into each of the N faulty circuits which are then run sequentially while comparing circuit outputs to the fault free circuit. An example

```
# the fault free circuit
for x = 1 to v
  do
  apply vector x;
  record fault free circuit outputs;
  done;

# N faulted circuits
for y = 1 to n
  do
      inject fault y;
      for z = 1 to v;
          do
          apply vector z;
          if faulty outputs != fault free outputs
              then
              report fault detected;
              next y;
          else
              report fault undetected;
          done;
    done;
report final fault coverage;
```

Figure 7. Pseudo code for a serial fault algorithm

of a sequential fault simulation algorithm is shown in figure 7, where n is the number of faults and v is the number of fault detection vectors.

The simulation time, for a serial fault simulation, tends to increase linearly with respect to the number of faults. Also the fact that the faulty circuit may differ only slightly from the fault free circuit indicates repeated simulation of many identical events. These two facts make CPU time, and thus cost, very expensive for this type of fault simulation. The main redeeming quality that this type of fault simulation has is very low memory usage. Only the true response needs to be stored in memory as the faulty response is generated and compared at run time. Any further simulation can be terminated, for a given fault, whenever a difference is detected from the stored response.

3.2 Parallel fault simulation

Parallel simulators are the next logical step after serial fault simulators. The parallel simulators take advantage of the fact that most machines perform logic word operations, typically with a 32 bit word. If the simulator considers logic signals to be one of two values, either high or low, then each bit in the word can be used to simulate one distinct fault. Theoretically a speedup of a factor of 32 can be achieved over serial simulation. This factor is typically not attainable however. As faults are detected they are dropped from the fault simulation. Therefore, any particular word contains much less than 32 faults as the simulation progresses. The effective speedup is thus greatly reduced. The benefit in speed-

up obtained by the ability to describe each fault as a logic high or logic low can also be a detriment in fault simulation. Many faults will manifest themselves as some logic value between a logic high or a logic low. This makes switch-level fault simulation impossible, in a parallel simulator, as most switch-level simulators express these intermediate values as unknowns.

3.3 Differential fault simulation

Differential fault simulation relies on the fact that each faulty circuit differs only by the inserted fault. In this type of simulator, a fault is inserted into a circuit model and then simulated. This fault is then removed and the next fault is inserted into the same circuit model. Then only the differences are simulated and propagated as far as possible. This process continues for each consecutive fault, simulating only the differences which are generally few. When the level of fault propagation is small this method can obtain significant speedup over serial fault simulation. However, the drawback to differential fault simulation is the extreme care which must be used in the fault list ordering. Fault propagation must be minimized so that successive differences, and thus the amount of simulation, is small. To achieve this objective, consecutive faults should be as close as possible to each other. This objective may be extremely hard to achieve for very large VLSI circuits.

3.4 Concurrent fault simulation

The predominant type of fault simulator today is the concurrent fault simulator. This

technique relies heavily on dynamic fault lists which are kept for each circuit node. Similar to the differential fault simulator, this type of fault simulation also takes advantage of the similarity of faulty circuits to reduce the amount of the simulation needed.

The main difference between the two types of simulators is that a concurrent simulator

will simulate the differences between the "good" circuit and the faulty circuit while a differential simulator simulates the differences between two consecutive faulty circuits. Concurrent fault simulation allows many faults to be simulated simultaneously with the limiting factor being the memory capacity of the machine that is running the simulator. With the cost of memory devices decreasing, very rapidly in recent years, this is becoming less of a limit. Many workstations can now be configured with 64 and even 128 Mbytes of resident memory space.

The basic concept behind concurrent fault simulation is to simulate only what needs to be simulated. If a particular gate or block, of a given faulty circuit, is guaranteed to have the same behavior as the fault free circuit, then the results of the fault free simulation will be used instead. This saves CPU time by not performing redundant simulations and reduces the amount of memory needed to store the faulty circuit(s) results. In addition, a gate or block need only be simulated if a given fault has propagated to this gate. CPU time may be reduced further by only performing a single evaluation for gates which have iden-

tical inputs among many faulty circuits. Combining all of these techniques can yield a significant reduction in CPU simulation time. This is indicated in Figure 8, where typical CPU times of serial fault simulation are compared to CPU times of concurrent simulation[14]. The initially higher CPU usage of the concurrent fault simulator is due to the

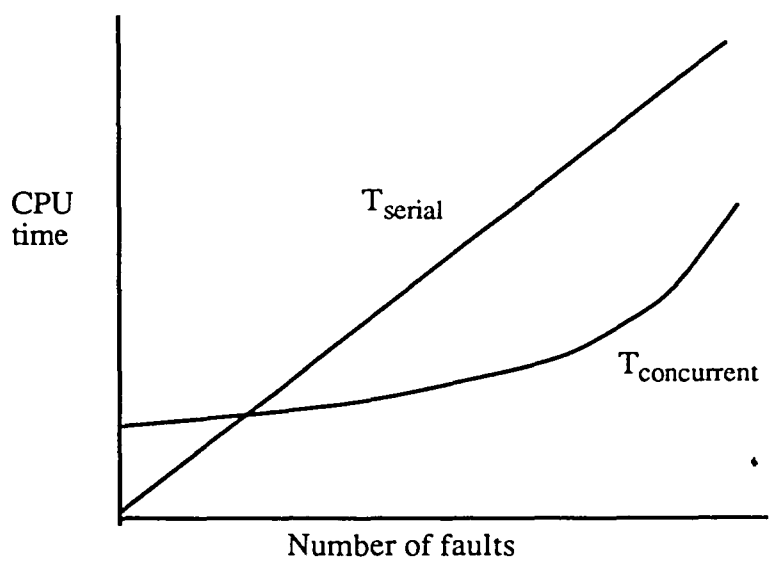


Figure 8. CPU fault simulation time versus number of faults

overhead of setting up fault lists, searching and sorting these lists, fault partitioning and dynamic memory management.

Much research has been done on multi-pass concurrent fault simulators that indicate excellent performance[14]. One simulator, in particular, was developed at AT&T which

incorporated a fixed memory storage capability for use on their Microprogrammable Accelerator for Rapid Simulation (MARS) systems[14,15]. The pseudo code, shown in Figure 9, gives an idea of the program flow for a multi-pass concurrent fault simulator.

```
pass_number = 1
vec_cnt = v
do begin
    insert next n faults
    initialize simulator
        for x from 1 to v
            advance simulation time
            schedule fault-free events
            schedule faulty event
                if fault-free events differ from faulty events
                    converge events and diverge, if necessary
                    simulate
                    If time to strobe
                        compare fault-free and faulty outputs
                    if faulty output != fault free output
                        drop detected fault(s)
                        converge detected faults
                    if all n faults detected or (x = v)
    pass_number++
done
continue until all faults processed
```

Figure 9. Pseudo code for a multi-pass concurrent fault algorithm

With this type of an algorithm, concurrent fault simulation is still being used. The difference in a multi-pass algorithm is that the total fault list is partitioned into many smaller lists of n faults each. It can be guaranteed that the available memory will never be exceeded by only simulating n faults, with each pass of the simulator. Of course, the fault-free simulation must be repeated for each of the partitions so there is some waste of memory resources in this type of an algorithm.

The primary drawback to concurrent fault simulation is its extensive memory usage. These types of simulators can be very efficient, however, the manipulation of fault lists requires a large performance overhead. This overhead can be excessive for very large VLSI circuits. Additional time may also be required for partitioning, if the available memory capacity is exceeded. Partitioning involves breaking the complete fault list up into multiple smaller simulation runs which will not exceed the resident machines memory capacity. However, the partitioning of the fault list itself needs additional memory, as the good circuit fault simulation must be performed for each partition. Due to the dynamic nature of the memory usage it is very difficult to predict when and how much partitioning will be necessary. Many times the partitioning is done by trial and error methods. A comparison of typical memory usage for concurrent simulation versus that of serial or sequential simulation, shown in Figure 10, graphically depicts concurrent fault simulations main

drawback.

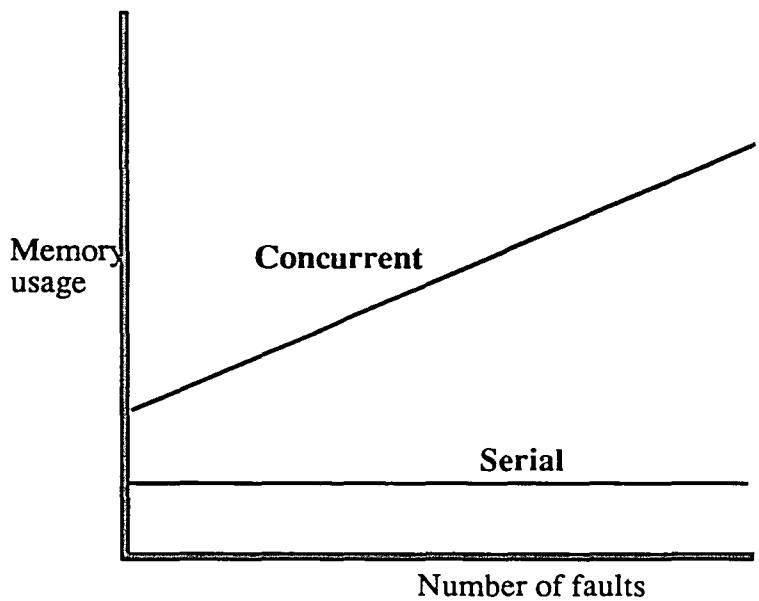


Figure 10. Simulation memory usage versus number of faults

3.5 IDDQ monitoring

A final type of fault testing, although not simulation based, is IDDQ monitoring; otherwise known as current testing. This type of fault testing is of prime interest in the design and testing of sequential circuits. The fault testing techniques for combinational circuits have been well researched and very high coverage is obtainable for node stuck-at faults[10]. IDDQ testing has shown the potential to increase a circuit's fault coverage, including fault classes other than stuck-at's, with no performance or area cost to the circuit design.

The concept of IDDQ monitoring is based upon the fact that most CMOS devices are designed to draw nano-ampere range quiescent currents. Current testing looks for excessive supply current by monitoring both VDD and VSS power pins. Tests are generated such that any existent faults will cause higher than expected supply current to be drawn by the device. Many faults, such as the bridging fault and transistor stuck-on, can activate low impedance paths between opposing device power supplies. These faults can be detected, by specialized test equipment, due to higher than normal steady state currents. The supply current can be monitored either by special on chip sensors, or very sensitive test equipment external to the device under test. This type of testing can be much faster than other widely accepted fault simulation techniques as the faults need not be propagated to primary circuit outputs. Figure 11 indicates the supply current for a simple CMOS logic gate, both with and without a transistor stuck-on fault.

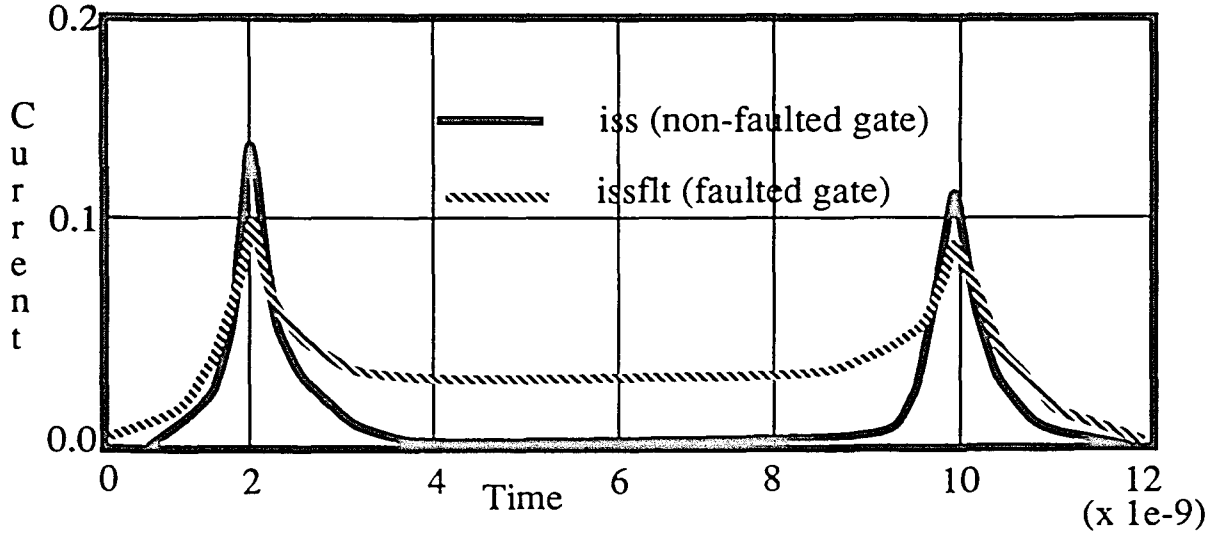


Figure 11. CMOS logic gate current measurements

It is apparent that the faulty circuit draws high supply current between switching transients whereas the fault-free circuit draws little, if any, steady state current. The primary drawback in the IDDQ testing technique is the difficulty in test generation. Great care must be exercised to get the logic in a state where these excess currents can be detected.

The fault simulator presented in this thesis uses a sequential or serial type algorithm. Ease of implementation was again a driving factor in this decision. It is our contention that the speed gained by using functional-level and switch-level logic simulators would far outweigh the reduction in speed by using a serial, rather than a parallel or a concurrent fault simulator. Adding in more parallelism, to the mixed-mode fault simulator, is discussed in section 6.1 under the heading of future work.

Chapter 4. Logic simulators

Use of a logic simulator is a standard practice when performing fault analysis on a given semiconductor circuit. As with the many types of fault simulators, there are also many types of logic simulators. Three of the primary types of logic simulators fall into the functional-level, gate-level and switch-level categories. Each of these types vary in degree of accuracy, speed and simplicity of the circuit model. The first types of logic simulators were typically of the gate-level variety. As technologies decrease in feature size and VLSI becomes more prevalent, these types of logic simulators are proving to be too cumbersome. The circuit model lacks the needed accuracy and the speed is too slow for very large designs. These facts led to the development of functional-level simulators for their speed, and switch level simulators for their speed and accuracy.

4.1 Gate-level logic simulation

Gate-level logic simulators have been the standard simulator of choice for many years. They model circuits as groups of gates such as AND's, OR's, etc., connected by, generally, delay free interconnects. Any delays are generally associated with the logic gates themselves. A fair amount of inaccuracy is indicated by this fact alone. It is well known that much of the propagation delay from one point to another, in a logic circuit path, is due to routing parasitics. An additional area of some inaccuracy comes from the fact that many

gate-level simulators use a two valued logic model. What is meant by this is that all inputs or outputs of the gates under simulation can be at logic "0" or logic "1". This model does not account for unknown node values or node voltage values which are somewhere between a logic "1" and the logic "0" level.

The circuit logic is simulated by performing an evaluation of a gates inputs and placing the corresponding value on its output. This output becomes the input of the next gate and so on. Due to this ripple type effect, care must be taken not to evaluate a gate with inputs that depend on the output of a gate not yet evaluated. It can be seen that the number of evaluations can become quite large and time consuming for this type of logic simulator. There is also a fair amount of accuracy lost due to the previously mentioned lack of wiring parasitics.

4.2 Functional-level logic simulation

Functional-level simulators are an attempt to far exceed the speed of gate-level simulators with only a modest loss of accuracy. These types of logic simulators are generally used at the system integration level or for VLSI devices. For a VLSI chip, a functional simulator describes each of the sub-blocks and how they communicate with each other. Each of the sub-blocks is described by a set of Boolean equations for its outputs. All inputs and state variables for the sub-block must also be described in this manner. The structure of the functional level simulator closely matches the real physical structure of the chip. For in-

stance, an ALU sub-block of a VLSI device might be a C function which contains an adder and a subtracter function call, where the addition and subtraction algorithm is described in detail. Generally these functional-level simulators are at a very high level of abstraction, such as a C program or at the register transfer level. It is because of the high level of abstraction and lack of accuracy that this type of simulator is generally not applicable to fault simulation. At the expense of some amount of speed, though, a rather fine level of accuracy can be encoded into a functional-level simulator[16].

4.3 Switch-level logic simulation

Switch-level simulators were developed in the early to mid-80's. These types of logic simulators became popular shortly after R. Bryant introduced the switch-level model for transistors in the early 1980's[18]. His switch-level simulator, MOSSIM, was unique in that it allowed circuits to be modelled at the transistor level rather than gate-level, as in a gate-level simulator. Since that time, switch level simulation has become an established method of verifying the steady-state behavior of circuits and many papers have been published on this topic[17,18,19].

In the switch-level model, a circuit is described as a set of nodes all interconnected by transistors or switches. Typically the three valued model is implemented in which each transistor is in the on, off or unknown state. For an NMOS transistor this would be equivalent to a logic high gate voltage, a logic low gate voltage or an X gate voltage respective-

ly. Depending upon the W/L (width/length) ratio of the transistor, there will also be an associated integer which represents its strength, in terms of conductance. Where greater accuracy is needed, a nodal capacitance can also be indicated for each of the circuit nodes. A switch-level simulator takes the network of varying strength switches, along with the nodal capacitances, and computes the steady state circuit behavior by solving groups of equations of the type $y = Ax + b$.

Switch-level simulators have many advantages that include superior speed over gate-level simulators. The individual transistor nature of a switch-level simulator also allows for many of the internal gate fault models to be used in this type of a logic simulator. Many circuit phenomenon such as charge sharing, ratioed logic and bidirectional signal flow can be dealt with in the switch-level model. Multiple opposing signals driving the same node to an unknown state, X, is also an advantage over the two valued logic model. Perhaps the greatest benefit of switch-level simulators is their ability to simulate full custom logic which may or may not be realized in any commonly recognizable logic gate formations.

Many of the original ideas of Bryant were incorporated in the switch-level SOISIM simulator which was implemented at AT&T Bell Laboratories[20]. There were additional novel concepts incorporated into SOISIM which increased its accuracy and speed. During the circuit compilation stage any well known logic structures are replaced with their functional equivalent. This would include many of the static combinational logic gates leaving

only the remaining gates to be simulated at the switch-level. A typical quoted number for the portion of a circuit which can be matched, to a given library of structures, is in the 85 percentage range. Another technique incorporated in SOISIM is the idea of partial ordering of strengths to resolve opposing signal conflicts. The authors claim that using this technique, as opposed to the traditional total strength ordering, simplified circuit modelling and also increased SOISIM's accuracy[17]. This switch-level simulator, SOISIM, has been used on many VLSI designs, at AT&T Bell Laboratories, with very satisfactory results.

This thesis presents a unique combination of both a functional-level logic simulator, HFSIM, and the switch-level logic simulator, SOISIM. Fault simulation is performed using both types of simulators with the majority of the faulty model being at the functional-level. Only the sub-block of the whole circuit which contains the faulty node is simulated at the switch-level. The mechanics of this will become more evident in chapter 5 where the fault simulator implementation is discussed.

Chapter 5. The mixed-mode fault simulator

With the size of VLSI MOS designs increasing by leaps and bounds, a need was felt for a fault simulator which could be used to analyze these extremely large circuits in a timely and cost effective manner. It is our contention that the finer qualities of functional-level simulation and switch-level simulation must be combined to perform this function. The speed of functional-level simulation must be combined with the accuracy of transistor switch-level simulations for a complete solution. The additional ability to incorporate multiple fault models into a single simulator is also highly desirable. This chapter presents a fault simulator implementation which uses a combination of functional-level simulation and switch-level simulation to achieve the above objectives. Described herein is the marriage of a functional-level simulator, written in the C programming language, with a switch-level simulator, to perform fast and accurate fault simulation. The initial implementation is a serial fault simulator using the stuck-at fault model. In the next chapter the ease of incorporating other fault models will be discussed, along with other future directions.

5.1 The functional level simulator

In VLSI chip design such as microprocessor design, it is very common to write a functional-level simulator. The work performed for this thesis was based on a microprocessor

functional-level simulator, called HFSIM, which has evolved over a number of years[16]. This simulator is written in the C programming language and is a clock phase accurate and algorithm accurate model of any functional events which may occur in a microprocessor. As new or improved functions were added to the design, they could be tested quickly and accurately under various benchmark programs and test conditions. Each individual sub-block of the design is a C function which is called, in the main program, during the appropriate clock phase. Inputs and outputs are specified, for each of the sub-blocks, in a corresponding C header file. New input, output and ioput (bi-directional signals) data types are defined in the main header file, as shown in figure 12 below. The new data types clearly show which signals are output from the sub-block and which signals are input to the sub-block. These variable types are used for all inputs and outputs to a given sub-block of the overall design. Single bit signals would fall into the input, output, or ioput category. Multiple bit signals, such as data busses, would fall into the inputi, outputi, or ioputi category where an ioputi is a multiple bit bi-directional signal.

```
#define inputi extern int
#define input extern unsigned char
#define outputi extern int
#define output extern unsigned char
#define ioputi extern int
#define ioput extern unsigned char
```

Figure 12. Data types describing a blocks inputs, outputs and bi-directional signals

The interconnection of the entire design at the sub-block level can be determined by studying the complete set of header files. Algorithms for addition, subtraction, multiplication and even cache replacement policies are all program coded in each sub-block, depending on its functional intent. These algorithms are implemented at a very fine level with logical expressions for each sub-block signal that is used to define the particular algorithm. Each signal is evaluated in its proper clock phase. Figure 13 gives an overly simplified example of how a read and a write operation for a memory sub-block might appear in this simulator.

```
cache_mem()
{
        if (clk1) {
                addr = newaddr;     /* memory access address */
                enable = enmem;     /* memory enabled when logic 1 */
                rw = readwrite;     /* active high read signal */
        }
        if (clk2)  {            /* memory read in clock phase 2 */
                  if ( rw && enable ) {
                  datao = mem[addr];
                }
        }
        if (clk4)  {            /* memory written in clock phase 4 */
                if ( !rw && enable ) {
                    mem[addr] = datai;
                }
        }
}
```

Figure 13. Memory sub-block functional-level definition

The C header file which would accompany the functional sub-block in figure 13 is shown in Figure 14. Notice that the data in and data out signals can be up to 32 bit busses as they are defined as integers. Each and every block in the simulator is organized in the same manner as this example.

```
/* cache_mem.h, header file for cache_mem functional sub-block definition */

input clk1;          /* phase 1 clock */
input clk2;          /* phase 2 clock */
input clk4;          /* phase 4 clock */
input newaddr;       /* 00,04 */     /* new 5 bit address from control block */
input enmem;         /* memory enable signal */
input readwrite;     /* memory read/write signal, active low write */
inputi datai;        /* 00,15 */   /* 16 bit wide data in bus */

outputi datao;       /* 00,15 */    /* 16 bit wide data out bus */
```

Figure 14. C program header file for cache_mem function

As mentioned previously, this functional-level simulator, HFSIM, is a clock phase accurate representation of all activity that occurs in the microprocessor. There are roughly 20,000 lines of C program code to implement the entire design. There is also a simulated memory model such that assembled or compiled programs can be downloaded and then run on the microprocessor functional model. Another very important extension to this

functional-level simulator is an integrated switch-level simulator. The integration of a switch-level simulator allows any blocks of the overall chip model to be simulated at a transistor level rather than a functional level.

5.2 Integrated switch-level simulator

The switch-level simulator, SOISIM, which was described in section 4.3 of this thesis, has unique characteristics compared to other switch-level simulators. This simulator can be executed either in stand alone mode or through a program level interface. With the use of the program level interface, any block or group of blocks can have its functional-level description replaced by a switch-level simulation. The result is an extremely efficient mixed-mode simulation capability for very large chip designs.

When running a particular block at the switch-level, this block's C function is replaced, in HFSIM, by another specialized C function. The specialized C function replaces the former logical expressions with SOISIM program level library functions. These functions bind the appropriate C variable at the blocks inputs to the associated transistor gate signals in the blocks transistor netlist. A switch-level simulation is then performed upon the transistor netlist. Finally, signal values resulting from the switch-level simulation are mapped back to the appropriate C variable in the functional-level portion of the full chip model. This can be done with groups of blocks or even the full chip by merely combining the appropriate C header files and sub-block transistor netlists.

The specialized C function is automatically generated by a program which takes as in-

put the block's C header file and the I/O statements from the transistor netlist. If a multiple bit signal is defined as an integer, in HFSIM, this program takes care of binding all of the used bits, out of a possible 32 bits, to the individual wires in the netlist. A group of two or more related SOISIM output signals are also compressed into a C integer variable at the completion of the switch-level simulation. Appendix A shows an example of the specialized C function for the memory sub-block depicted in Figure 13.

The combination of the functional-level and switch-level simulators, described in section 5.1 and 5.2, is what gives this fault simulator its speed. Functional-level simulations presently run on a SUN Sparcstation roughly 11,000 clock cycles per minute. This is with none of the sub-blocks simulated at switch-level. For comparison, the mixed-mode simulator would run about 5,000 cycles per minute with a sub-block of roughly 650 gates, including the simulation at the switch-level.

5.3 Program flow of the mixed-mode fault simulator

This fault simulator was specifically implemented with built in flexibility such that the fault coverage can be determined in various manners. The size of the block being switch-level simulated can be from very small, 300-400 gate sub-blocks, all the way up to the full chip. With this ability the overall fault coverage can be arrived at by simulating many small blocks or one very large block. The advantage of simulating many smaller blocks is that the simulation work can be subdivided amongst many machines or processors thereby

achieving more parallelism, and thus increased simulation speed. Simulating many sub-blocks, rather than a full chip design, can also quickly give a clearer picture of where there may be trouble spots of low fault coverage. On the other hand, one may not have the resources to simulate many small sub-blocks. The flexibility is built into this fault simulator, such that a full chip design can also be fault simulated.

The operation of this fault simulator can be broken down into four main phases. These being the collapsed fault list generation phase, the faulty model generation phase, the fault simulation phase, and the data postprocessing phase. As with most other fault simulators the time consuming phase is the fault simulation itself. A number of programs were written to perform the various functions of each phase. Table 1 lists the program names, the language used and a brief description of the programs function.

Table 1: Mixed-mode fault simulator sub-programs

Program name	Programing language	Program description
FLTSIM	C	Reads hierarchical circuit netlist and outputs a flattened circuit netlist.
PUTFLT	C	Reads the fault list and performs the indicated node faulting operations on the fault-free netlist.
BVALDIF	C	Compares the "good" circuit response file with a faulty circuit response file.
FLTCVG	AWK	Processes fault binning files and reports fault coverage and other simulation statistics.

The various phases of this fault simulator are depicted in a high level flowchart in Appendix B. As can be seen, the flowchart depicts a sequential or serial fault simulator. Any number of fault vectors can be simulated. The size of the block, to be simulated, is also variable. Larger size blocks can be simulated by merely combining the necessary C header files to get the correct input/output list, and then linking the associated netlists together. The initial approach was kept simple to verify the feasibility of the overall simulator. Methods are presented, later in this thesis, to increase the speed through such ideas as multiple workstation parallelism and "on the fly" circuit output comparisons.

5.4 Preprocessing transistor netlist

Blocks of the microprocessor described by the functional-level simulator, HFSIM, will be discussed in the remainder of this chapter, as this simulator was primarily developed for fault simulation of this design. The available transistor netlist descriptions were obtained from a schematic capture tool which outputs a hierarchical netlist. This format proved extremely difficult to find and insert faults into the hierarchical netlist. The entire hierarchy of the device had to be traversed to insert a single stuck-at fault. Traversing the hierarchy also proved to be too time consuming and memory intensive. An alternative to this approach was to write a simple program to flatten the hierarchy and output a gate level netlist. A flattened connectivity netlist proved to be optimal for fault insertion and simulation.

The flattening program, FLTSIM, was written in the C programming language. This

program enabled any of the blocks of this design to be modified from a hierarchical netlist to a flattened gate-level netlist in a matter of seconds. Table 2 illustrates the sizes and conversion times for a few of the control sub-blocks of this microprocessor.

Table 2: Hierarchical to flattened netlist conversion times

Block name	Gate count	Conversion time (sec.)
eualu	368	6.4
euc	372	6.7
euio	836	12.4
euop	407	6.6
eupc	510	7.7

This conversion program was written so that the flattened netlist could be output in a variety of standard netlist formats. Input and output signals were reordered such that output signals always follow the complete list of inputs. Benefits were gained by this ordering during the fault insertion portion of the preprocessing phase. Gate terminals also kept their hierarchical names for easier search and locate routines. The flattened netlist format allowed the gate search and fault insertion tasks to proceed much more rapidly than in the hierarchical netlist format.

5.5 Fault list generation

Fault list generation, collapsing and random sampling were all accomplished using a

front-end tool developed for an already existent fault simulator within AT&T Bell Laboratories. This tool, FSIM, handled these tasks very efficiently and therefore there was no need to reintroduce this capability. The fault list generation is automated but is limited to stuck-at-0 and stuck-at-1 faults at the present time. Automating this portion of any fault simulator is essential. Manual generation of fault lists, is both a tedious and time consuming process which is generally not possible for VLSI designs.

FSIM takes, as input, the compiled gate-level netlist and can output a full fault list, a collapsed fault list, or some random sampling of either the full or collapsed fault list. Initially, stuck-at faults are generated for each node in the circuit. This list is then pruned down by fault collapsing and other user supplied options. Command line options allow a seed value to start the random sampling and a specification of the complete sample size. Table 3 shows some fault list statistics for the 5 microprocessor control blocks indicated in Table 2.

Table 3: Fault list generation statistics

Block name	Gate count	Total faults	Collapsed faults	Generation time (sec.)
eualu	368	3138	1856	36.4
euc	372	3662	1950	37.2
euio	836	7046	3554	66.6
euop	407	3162	1646	35.2
eupc	510	3887	2251	38.4

FSIM also provides the option to "unfault". This allows any individual faults to be removed from the generated fault list. A few faults may cause an inordinate amount of circuit activity and the fault simulation may take an unacceptable amount of time to finish. These, so called oscillation faults, can be removed from the generated fault list to enable the simulation to complete and then be examined more closely at a later time.

A modification was made to the code used to perform fault collapsing. This code was changed to put more weight on the selection of output faults in any group of equivalent faults. This was done to improve the speed of the fault insertion routines. For an input fault, the particular faulty gate is first located and then only the affected input is selected and tied to the given stuck-at logic level. In the output fault case, the netlist is parsed and wherever the faulty node appears it is tied to the appropriate stuck-at logic level. Output faults, therefore, can be inserted slightly more quickly than an input fault.

5.6 Fault insertion routine

The fault insertion portion of the presented fault simulator is written in the C programming language. This program, PUTFLT, takes as input consecutive lines from the previously generated fault list and performs the indicated node faulting to a fault free circuit netlist. Since this simulator, presently, only supports the stuck-at fault model, a fault can be inserted by the connection of the faulty node to the power supply or ground. For output faults, the indicated node name is replaced globally with the stuck-at logic level. This can be done because theoretically an output stuck-at fault should affect every gate that it fans into. An input fault is a little more difficult to handle. For these types of faults the gate-

level netlist must first be scanned for the affected gate. Then the gates input /output list is scanned to select the proper input. This single input node is then connected to the indicated stuck-at logic level.

A problem that arose almost immediately was the faulting of a sub-blocks primary inputs and outputs. Stuck-at-1 faults replaced the indicated node with VDD and stuck-at-0 faulted nodes were replaced with VSS. When this was done on a block primary input or output, the previous signal name no longer existed. This had dire consequences when the special C function, presented in section 5.2, attempted to bind the functional-level portion of the chip to the block represented at the switch-level. The situation was rectified as shown in Figure 15.

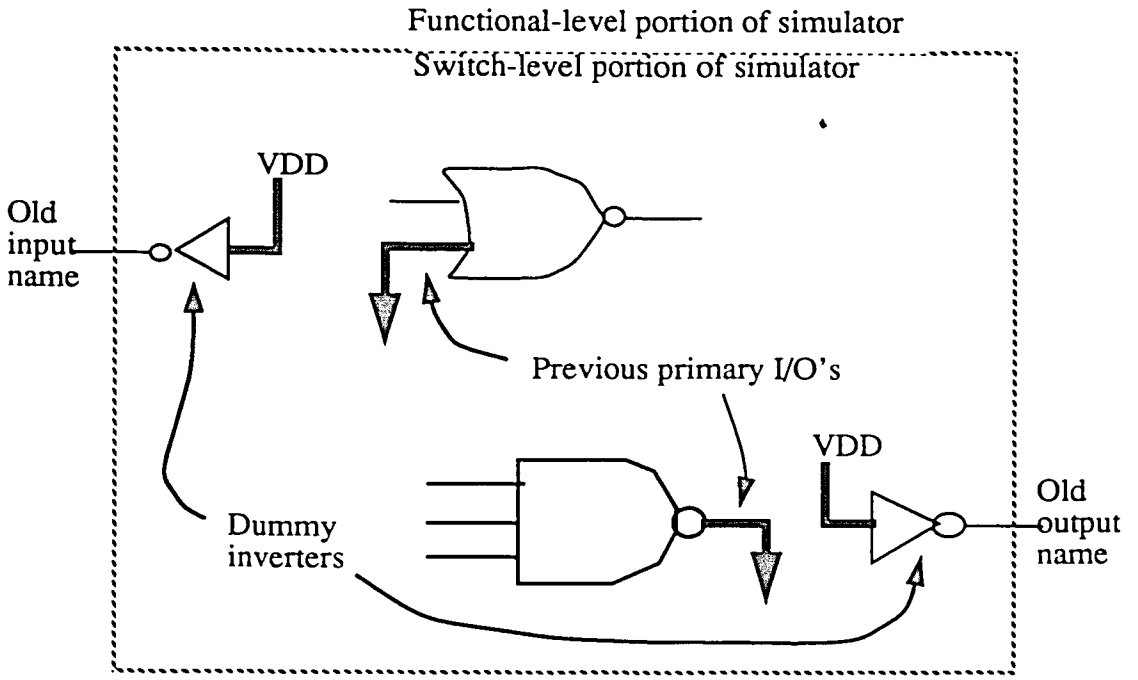


Figure 15. PUTFLT Input/Output binding solution

The example shows a stuck-at-0 input fault and a stuck-at-0 output fault, both of which are primary input or output signals. In both cases a dummy inverter is inserted into the netlist with its output named as the old primary input or output signal name. The input or output fault is then inserted as previously explained. The dummy inverters outputs allow the functional-level simulator to bind to terminals within the netlist without affecting the switch-level results. The inverter for the output fault passes a logic "0" to the functional-level simulator as a stuck-at-0 fault on the NAND gate output would. The inverter at the input fault location merely gives the signal from the functional-level simulator a binding point.

The PUTFLT fault insertion routine is extremely fast, with each fault typically taking less than half a second to insert. This is for a 300 to 400 gate logic block. Fault insertion times are, of course, dependent on the size of the netlist and the number of nodes to be faulted as shown in table 2. However, even the EUIO block (see table 2), with over 800 gates averaged less than a second to insert each fault. The time to insert all 3162 faults for the EUOP block (see table 2), into separate faulty netlists took less than 1 hour of CPU time. It is obvious that generating the faulty circuit models "on the fly" is not a problem.

5.7 Fault simulation process

Once the preprocessing work has been completed the CPU intensive simulations begin. The "good" simulation is first run using a given set of fault test vectors. This simula-

tion is run using a purely functional-level simulator for maximum speed and the circuit response is stored in a binary file. The first fault is then inserted into a fault free sub-block netlist description, by the PUTFLT program. At this point, the sub-blocks C functions are replaced, in HFSIM, by the specialized C functions, as described in section 5.2. This replacement need only be done once as the I/O signals, for the sub-block, will always have the same names for each of the fault simulation runs. The specialized C function will also call the same faulty circuit's name for each sequential fault simulation. This file will be overwritten for each consecutive fault so that a different faulty circuit is used for each simulation. A combination of functional-level and switch-level simulation is then run for each consecutive fault. Only the sub-block, from which the fault list was derived, is simulated at the switch-level. The steady state responses from the switch-level simulation are passed to the functional-level portion of the faulty circuit model. These steady-state responses, from the switch-level simulation, may or may not propagate to the microprocessor's output pins. In either case, the responses at the microprocessors output pins are stored in another binary output file.

As each consecutive fault simulation is completed, a line by line comparison is performed between the "good" circuit response and the faulty circuit response files. Any difference is flagged as a detected fault and recorded in a detected fault binning file. Once flagged as detected, a fault is dropped from the total fault list. The fault simulation con-

tinues until all available fault vectors have completed. If at any time there is not a discernible difference between the two response files, then the fault is flagged as undetected and recorded in the undetected fault binning file.

The circuit response files are compared using a program specifically designed for this function. This program, BVALDIF, compares binary output files for the added speed over ASCII file comparisons. BVALDIF compares the "good" circuit response to the faulty circuit response up to a point specified by the user. The circuit responses can be compared for the entire simulation run if desired. This might be beneficial for a user to observe how the faulty response varies as the fault propagates through the design. An option is also provided so that the response comparison will cease after the first difference is detected. A header is output, by BVALDIF, listing each of the response signals in columnar format for easy viewing of which output response was faulty. There is also a waveform plotting capability built into the BVALDIF program so that the "good" and fault responses may be observed side by side.

The final phase of this fault simulator is the data postprocessing phase. Data manipulation is easily performed by a pattern-action type programming language such as AWK under the UNIX operating system. An AWK program, FLTCVG, was written to process the fault detected and fault undetected binning files. The statistics from these two files are used to report the fault coverage for the sub-block under test. The fault undetected binning

file is processed to generate a report of the faults which went undetected during this fault simulation run. This report is typically used to enhance the fault vector set such that these undetected faults will be sensitized by future simulation runs. Finally, a report is generated concerning statistics of the latest simulation run. This includes CPU time, vector count and memory usage.

The overall fault simulation process can be broken down into a number of activities. These activities, along with a brief description are shown in table 4 below.

Table 4: Fault simulation activity breakdown

Activity	Description of task
Flatten netlist *	Flatten hierarchical circuit netlist
Generate fault list *	Run FSIM to generate either a full or randomly sampled fault list
Build mixed-mode simulator *	Replace sub-block C function with specialized C binding function and recompile HFSIM.
Insert faults	Consecutively insert faults into fault-free netlist
Simulate "good" circuit *	Run full set of fault vectors on fault-free circuit model and store output response.
Simulate faulty circuit	Run full set of fault vectors on each consecutive faulty model and store output response.
Response comparison	Perform line by line comparison of fault-free circuit response and faulty circuit response files.
Data postprocessing *	Report final fault coverage and fault simulation run statistics.

These listed activities are performed for each simulation run. The activities which are indicated by a "*" are only performed once for each total simulation run. The other activities will be performed multiple times during the course of each full simulation run. A bar graph showing measured CPU times for each of the listed activities is depicted in Figure 16. This bar graph shows times representative of logic block the size of eualu, euc and euop (see table 2).

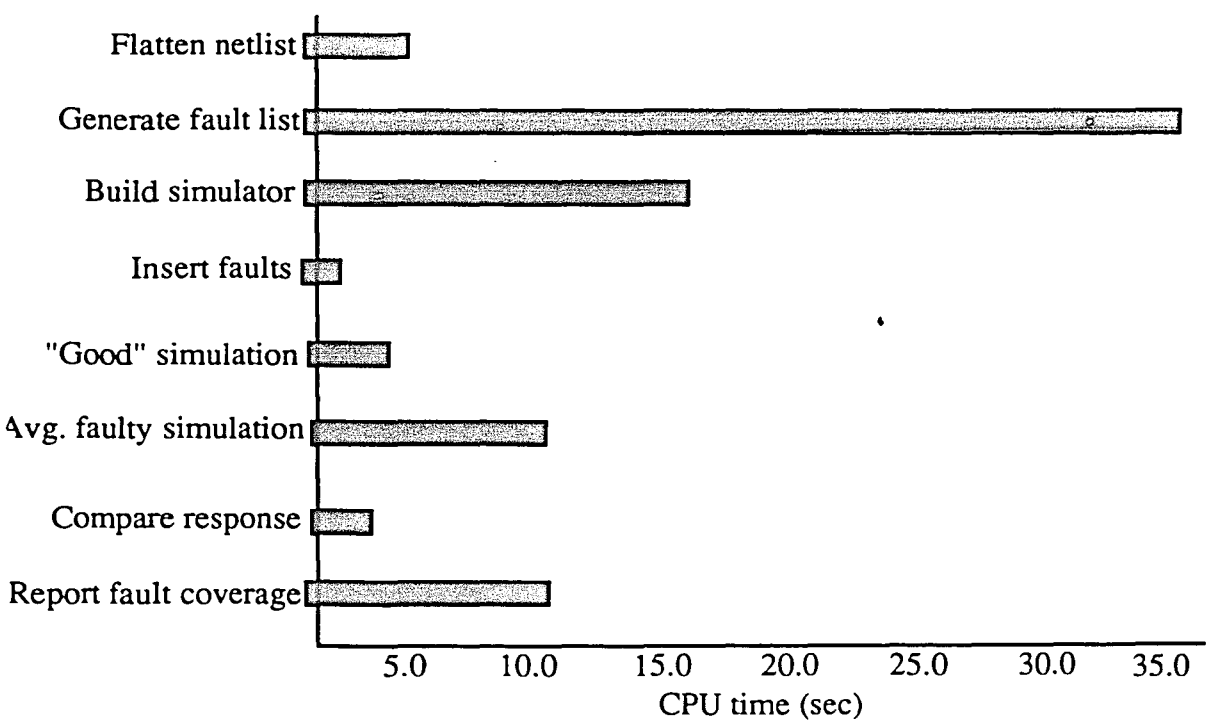


Figure 16. Fault Simulation activity time breakdown

Chapter 6: Future work and conclusions

The simulator, presented in this thesis, is the initial implementation of a mixed-mode fault simulator which is both fast and accurate. It is a simulator which uses the best of both functional-level and switch-level simulators to perform fault analysis of VLSI MOS circuits, such as microprocessors. The implementation of the simulator proved to work very well for the majority of the test vehicle sub-blocks, including full custom logic circuits. This first implementation was intentionally kept simple to prove that the concept would work. However, this fault simulator is far from perfect and there are many areas for improvement.

6.1 Future work

The speed and accuracy of the present fault simulator implementation are commendable. As in any design, however, there is still room for improvement. Some of the potential areas for improvement have been presented throughout this thesis and are listed again for completeness:

- Additional fault models
- Transistor-level rather than gate-level simulation
- Maintain netlist hierarchy
- Additional simulation parallelism
- "On the fly" output response comparison

The first two items on the list for future work are interrelated. Stuck-at fault models can rather easily be inserted into gate-level netlists. To expand the fault models that this simulator can handle would necessitate using transistor-level netlists. The program, FLTSIM, already has the capability to output flattened transistor-level netlists. Therefore, modifying the simulator to use this type of a netlist should not be a lot of work. The primary routine affected would be the fault insertion routine. This will have to be changed to parse transistor descriptions rather than gate descriptions.

Using a switch-level logic simulator enables the inclusion of fault models, other than the stuck-at model, quite easily. In the future, this simulator should be capable of handling any of the point defect models described in chapter 2. The transistor stuck-open and stuck-on models are easily implemented since all terminals of each transistor are accessible in a switch-level model. Delay faults and bridging faults can be modelled by adding the appropriate capacitance to, or between, specific nodes of the faulty circuit. Questions that need to be answered, before implementing these models, are numerous though. Is the percentage of real defects which cannot be modelled by the stuck-at model statistically significant? Is the fault coverage gained worth the loss in simulation speed? These are a few of the questions which must be answered before spending much time and effort to implement other fault models.

The third area of future work runs contrary to the first two but is an accepted need as

VLSI designs move towards ULSI designs. As the circuits become larger, more and more CAD tools are using hierarchical design methodologies. A hierarchical circuit connectivity list is much smaller in size and easier to manage than a flattened one. Flattened netlists are easier to manipulate but are very wasteful of memory resources. The same sized and configured gate can appear many times in a given design. This gate need only be represented once in a hierarchical netlist in contrast to being repeated many times in a flattened netlist. Ways are being looked at to maintain the design hierarchy without compromising the speed at which fault insertion occurs.

With the amount of Local Area Networking (LAN) which is seen today for workstations, it should be fairly easy to build more parallelism into this fault simulator. Each individual fault simulation does not take a great deal of time, but they are still sequential in nature. Means are being studied to run many fault simulations on remote SUN workstations, in parallel. There are also many machines, today, which have multiple processors, such as the MARS hardware accelerator[15]. Ways to utilize these multiple processors is another area of future work.

Another area of future work is "on the fly" output response comparison. The present simulator does a comparison only after both the "good" and the faulty simulations have completed. BVALDIF performs the comparison and is presently quite fast. If the comparison could be done while the faulty circuit is under simulation, then any further simulation

could be aborted once a difference is found. This is called "on the fly" comparison and can be very beneficial both in reducing the simulation time and comparison time. Ways to modify BVALDIF, to perform this function, are being studied.

6.2 Conclusions

The design of increasingly larger VLSI circuits is making it extremely difficult to maintain the quality level of outgoing products. Present day VLSI designs have orders of magnitude more devices than those of only a few short years ago. Integration of many devices onto a single piece of silicon, as is the trend, has increased the area and thus, the susceptibility of devices to fabrication defects. Many of the recent developments in clean rooms have been aimed at reducing lithography features rather than improving device reliability. Fault simulation is intended to detect any of the manufacturing defects which are present in the device. The increasing size and complexity of today's VLSI design, however, is putting an enormous burden on presently available fault simulation CAD tools. The demand for accurate and efficient fault simulation, of digital MOS circuits, is growing.

The intent of this thesis was to present a unique fault simulator implementation for VLSI digital MOS circuits. Presented was a simulator which uses a combined functional-level and switch-level approach to fault simulation. This methodology provides an accurate as well as efficient means for performing fault analysis of very large custom and standard cell circuit designs. In addition to the "classical" node stuck-at fault model, this

simulator has the ability to model many of the other proposed physical models of fabrication defects. This fault simulator can include models such as the transistor stuck-on, the transistor stuck-open along with the bridging and delay type fault models. The simulation CPU time achieved was faster than traditional gate-level fault simulators. The large number of implementable fault models also yields a far better fault coverage than strictly stuck-at fault simulators.

A wealth of information, concerning fault analysis, has been presented in this thesis. Some background material and a general introduction to the importance of fault analysis was covered in the first chapter. The cost of not providing adequate test coverage was shown at various levels of product assembly. A couple of means for measuring device quality were also discussed in the latter portion of chapter 1. Chapter 2 was used as an overview of many of the physical models which have been proposed to describe the numerous fabrication defects which can occur in a VLSI circuit. The transistor stuck-on, stuck-open, bridging and delay fault models were presented along with the "classical" stuck-at fault model. Chapter 3 dealt with a few of the more common types of fault simulators. Another type of fault detection, called IDDQ monitoring or current testing, was also presented in this chapter. Chapter 4 dealt with the underlying logic simulators used in fault simulation. The gate-level, functional-level and switch-level logic simulators are three of the most common types of logic simulators. In chapter 5, the actual implementation and

other issues about the new mixed-mode fault simulator were discussed. Programs within the fault simulator implementation to flatten hierarchical netlists (FLTSIM), generate fault lists(FSIM) and insert faults(PUTFLT) were also discussed in chapter 5. Finally, future work and methods to improve the speed and include additional fault models for this fault simulator were presented in section 6.1.

The fault simulator presented herein, along with techniques such as IDDQ monitoring, should be capable of delivering near total fault coverage for VLSI MOS designs. The ultimate fault coverage is, of course, dependent upon how well a set of fault vectors can be written to exercise the design. However, it has been shown that this fault simulator can take a given fault vector set and run them in a timely, cost efficient and accurate manner. The fact that full custom logic can be simulated without problems is an added bonus. Switch-level logic simulation makes this possible and also enables the fairly simple inclusion of additional fault models at a future time. The fault simulator, presented in this thesis, has proved to work very well with today's VLSI circuit designs. It also has room to grow and expand its abilities for tomorrows highly integrated and complex ULSI circuits.

References

[1]S. Seth, V. Agrawal, " Cutting Chip Testing Costs ", IEEE Spectrum, April 1985, pp 38-45

[2]D. Vir Das, S. Seth, P. Wagner, J. Anderson, V. Agrawal, "An Experimental Study on Reject Ratio Prediction for VLSI Circuits: Kokomo Revisited", Proc IEEE International Test Conference, March 1990

[3]T. Williams, N Brown, "Defect Level as a Function of Fault Coverage", IEEE Trans. Computers, Vol C-030, pp987-988, Dec 1981

[4]S. Seth, V. Agrawal, "Characterizing the LSI Yield Equation from Wafer Test Data", IEEE Trans. Computer Aided Design of Integrated Circuits and Systems, Vol CAD-3, pp 123-126, April 1984

[5]R. Wadsack, "Fault Coverage in Digital Integrated Circuits", B.S.T.J., Vol 57, pp 1475-1488, May-June 1978

[6]W. Maly, "Realistic Fault Modelling for VLSI Testing", SRC-CMU Research Center for CAD, Report CMUCAD-87-27, pp 1-8, June 1987

[7]J.Fergusun, M. Taylor, T. Larrabee, "Testing for Parametric Faults in Static CMOS Circuits", Proc. IEEE International Test Conference, pp 436-443, 1990

[8]W. Maly, "Modelling of Point Defect Related Yield Losses for CAD of VLSI Circuits", Proc of International Conference on Computer-Aided Design, Nov 1984

[9]C. Lo, H. Nham, J. Bose, "Algorithms for an Advanced Fault Simulator System in MOTIS", IEEE Trans. of Computer-Aided Design of Integrated Circuits and Systems, pp 232-240, March 1987

[10]R. Fritzemeier, J. Soden, R. Keith, K. Treece, C. Hawkins, "Increased CMOS IC Stuck-at Fault Coverage with Reduced IDDQ Test Sets", Proc . IEEE International Test Conference, pp 427-435, 1990

[11]T. Storey, W. Maly, J. Andrews, M. Miske, "Stuck Fault and Current Testing Comparison using CMOS Chip Test", Research report CMUCAD-

91-8, Feb 1991

[12]J. Fergusun, M. Taylor, "Comparison of IDDQ and Single Stuck-at 0 / 1 Testing Costs", University of CA Computer Engineering and Information Sciences, 1989

[13]J. Soden, C. Hawkins, R. Fritzemeier, J. Guth, "CMOS IC IDDQ Testing for the 1990's", IEEE VLSI Test Symposium, April 10, 1990

[14]P. Agrawal, V. Agrawal, K. Cheng, R. Tutundjian, "Fault Simulation in a Pipelined Multiprocessor System", Proc. International Test Conference, 1989

[15]P. Agrawal, "MARS: A Multiprocessor Based Programmable Accelerator", IEEE Design and Tets of Computers, Vol 4, pp 28-36, Oct 1987

[16]A. Berenbaum, R. Heaton" Functional Simulation for the CRISP Microprocessor", Proc of the IFIP TC 10/WG 10.5 International Conference on VLSI, pp 323-334, Aug 1987

[17]T. Szymanski, P. Agrawal, S. Robinson, "Automatic Modelling of Switch-level Networks Using Partial Orders", IEEE Trans. on Computer-Aided Design, Vol 9, No 7, pp 696-707, July 1990

[18]R. Bryant, "A Switch-level Model and Simulator for MOS Digital Systems", IEEE Trans. Computers, Feb 1984, pp 160-177

[19]I. Hajj, "A Path Algebra for Switch-level Simulation", IEEE Int. Conference on Computer-Aided Design, Santa Clara, CA, pp 153-155, November 1985

[20]R. Gadenz, W. Hayes, L. Bays, C. Chen, E. Fields, H. Moscovitz, T. Szymanski, "Post-layout verification for the WE DSP32 digital signal processor", IEEE Design Test, Vol 6, pp 56-66, Feb 1989

APPENDIX A.

This appendix lists the specialized C function for the cache memory depicted in figure 13. This special C function is used to bind the functional-level portion of the mixed-mode fault simulator to the switch-level portion.

```
#if sun
typedef struct {unsigned charb7:1, b6:1, b5:1, b4:1, b3:1, b2:1, b1:1, b0:1;} _Byte;
typedef struct {unsigned intb31:1, b30:1, b29:1, b28:1, b27:1, b26:1, b25:1, b24:1,
   b23:1, b22:1, b21:1, b20:1, b19:1, b18:1, b17:1, b16:1,
   b15:1, b14:1, b13:1, b12:1, b11:1, b10:1, b9:1, b8:1,
   b7:1, b6:1, b5:1, b4:1, b3:1, b2:1, b1:1, b0:1;} _Word;
#endif

extern char clk1, clk2, clk4, enmem, newaddr, readwrite;
extern int datai, datao;

static char    _L[37];

static char_in[] = "->";
static char_out[] = "<-";
static char_io[] = "<->";

static struct {
     char*a, *b, *c, *d;
} _binds[] = {
{ &clk1,_in,"clk1" },
{ &clk2,_in,"clk2" },
{ &clk4,_in,"clk4" },
{ &_L[0],_in,"newaddr00" },
{ &_L[1],_in,"newaddr01" },
{ &_L[2],_in,"newaddr02" },
{ &_L[3],_in,"newaddr03" },
{ &_L[4],_in,"newaddr04" },
{ &enmem,_in,"enmem" },
{ &readwrite,_in,"readwrite" },
{ &_L[5],_in,"datai00" },
{ &_L[6],_in,"datai01" },
{ &_L[7],_in,"datai02" },
```

```
{ &_L[8],_in,"datai03" },
{ &_L[9],_in,"datai04" },
{ &_L[10],_in,"datai05" },
{ &_L[11],_in,"datai06" },
{ &_L[12],_in,"datai07" },
{ &_L[13],_in,"datai08" },
{ &_L[14],_in,"datai09" },
{ &_L[15],_in,"datai10" },
{ &_L[16],_in,"datai11" },
{ &_L[17],_in,"datai12" },
{ &_L[18],_in,"datai13" },
{ &_L[19],_in,"datai14" },
{ &_L[20],_in,"datai15" },
{ &_L[21],_out,"datao00" },
{ &_L[22],_out,"datao01" },
{ &_L[23],_out,"datao02" },
{ &_L[24],_out,"datao03" },
{ &_L[25],_out,"datao04" },
{ &_L[26],_out,"datao05" },
{ &_L[27],_out,"datao06" },
{ &_L[28],_out,"datao07" },
{ &_L[29],_out,"datao08" },
{ &_L[30],_out,"datao09" },
{ &_L[31],_out,"datao10" },
{ &_L[32],_out,"datao11" },
{ &_L[33],_out,"datao12" },
{ &_L[34],_out,"datao13" },
{ &_L[35],_out,"datao14" },
{ &_L[36],_out,"datao15" },
};

static int   _nbinds = 42;

cache_mem()
{
static int   initdone;
     if (!initdone) {
          readckt ("cache_mem.bsim");
          Xmem()
          initdone = 1;
```

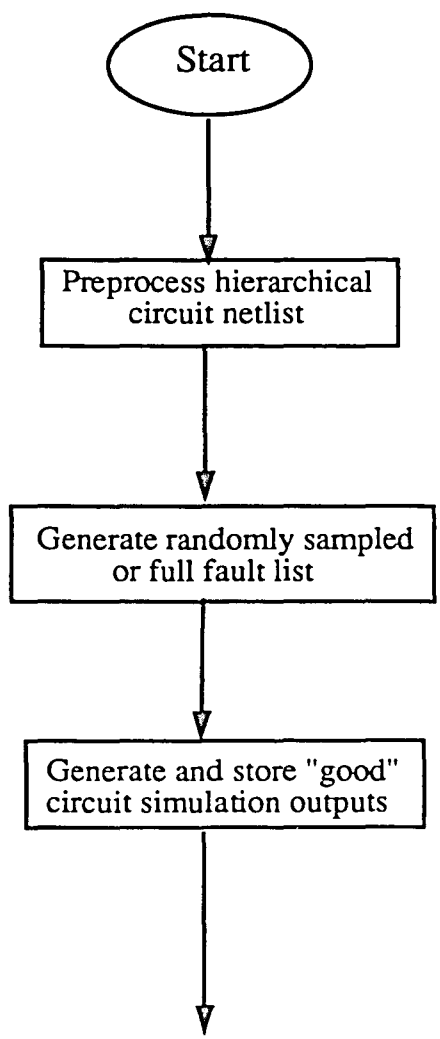
```
    }
    Ymem();
        settle();
    Zmem();
    return;
}

Xmem()
{
int   i;

    for (i = 0; i < _nbinds; i++)
            bind (_binds[i].a, _binds[i].b, _binds[i].c, _binds[i].d);
    return;
}


Ymem()
{
register long     _word;
_word = newaddr;
_L[0] = _word & 1;
_L[1] = _word >> 1 & 1;
_L[2] = _word >> 2 & 1;
_L[3] = _word >> 3 & 1;
_L[4] = _word >> 4 & 1;
_word = datai;
_L[5] = _word & 1;
_L[6] = _word >> 1 & 1;
_L[7] = _word >> 2 & 1;
_L[8] = _word >> 3 & 1;
_L[9] = _word >> 4 & 1;
_L[10] = _word >> 5 & 1;
_L[11] = _word >> 6 & 1;
_L[12] = _word >> 7 & 1;
_L[13] = _word >> 8 & 1;
_L[14] = _word >> 9 & 1;
_L[15] = _word >> 10 & 1;
_L[16] = _word >> 11 & 1;
_L[17] = _word >> 12 & 1;
```

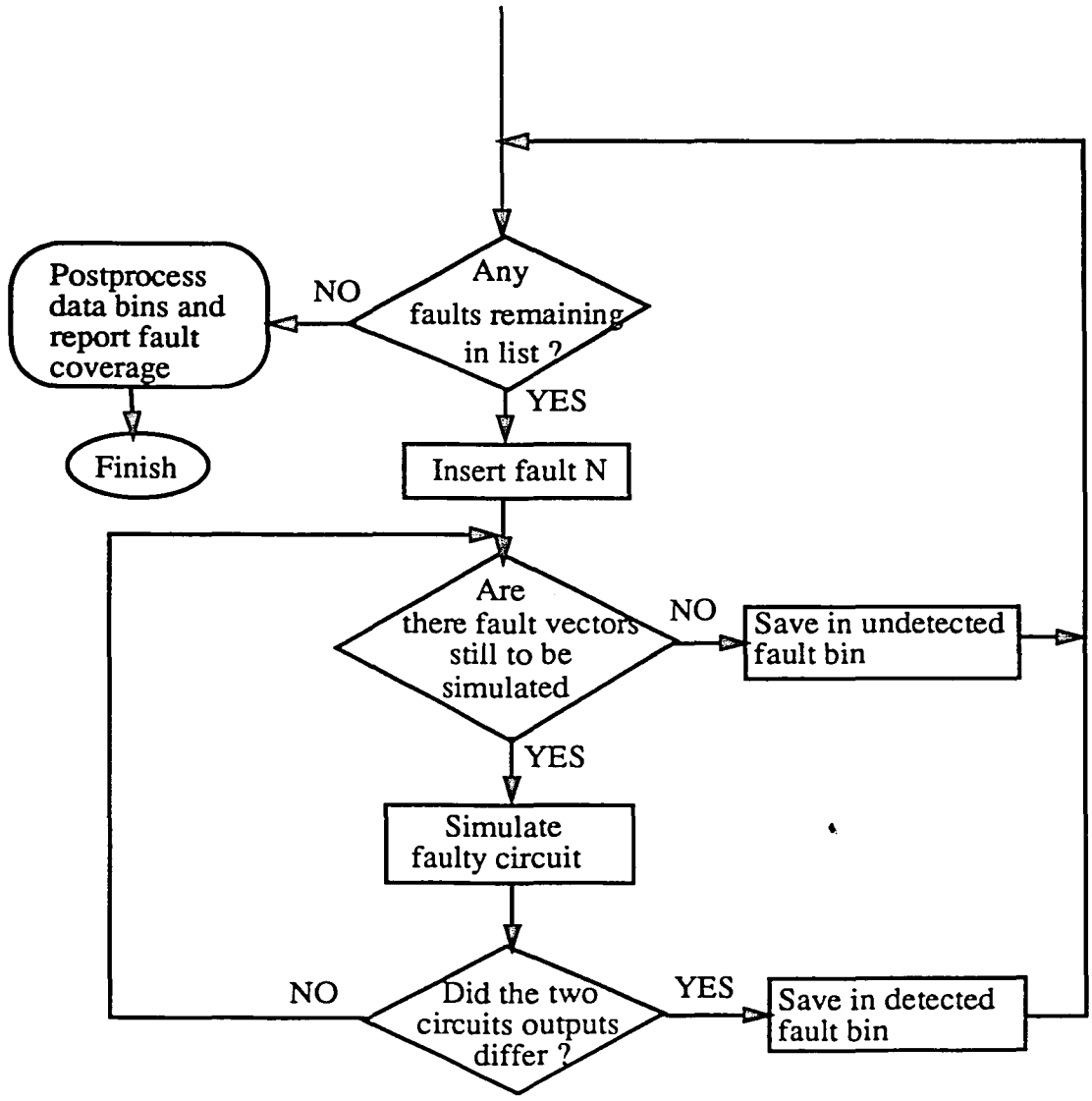
```
_L[18] = _word >> 13 & 1;
_L[19] = _word >> 14 & 1;
_L[20] = _word >> 15 & 1;
return;
}

Zmem()
{
register _Byte    *_byte;
register _Word    *_word;
_word = (_Word *) &datao;
_word->b0 = _L[21];
_word->b1 = _L[22];
_word->b2 = _L[23];
_word->b3 = _L[24];
_word->b4 = _L[25];
_word->b5 = _L[26];
_word->b6 = _L[27];
_word->b7 = _L[28];
_word->b8 = _L[29];
_word->b9 = _L[30];
_word->b10 = _L[31];
_word->b11 = _L[32];
_word->b12 = _L[33];
_word->b13 = _L[34];
_word->b14 = _L[35];
_word->b15 = _L[36];
return;
}
```

Specialized C function to bind functional simulator signals to SOISIM simulator signals

Appendix B.

Appendix B indicates the program flow of the mixed-mode fault simulator presented in this thesis. The diagram shows a sequential fault simulator and the various functions performed during a typical fault analysis.




Postprocess data bins and report fault coverage
NO
Any faults remaining in list ?
YES
Finish
Insert fault N
Are there fault vectors still to be simulated
NO
Save in undetected fault bin
YES
Simulate faulty circuit
NO
Did the two circuits outputs differ ?
YES
Save in detected fault bin

Biography

David A. Inglis received the B.S. degree in Electrical Engineering Technology from the New Jersey Institute of Technology in 1985. He is presently employed at AT&T Bell Laboratories, as a member of technical staff, in the Custom VLSI Design department located in Allentown, PA. His current areas of research interest are in microprocessor design and pen-point computing.

Previously, Mr. Inglis attended the State University of New York in Syracuse, N.Y. Here, he received the Associate Degree in Electrical Engineering Technology. This university was attended from 1978-1980.

Mr. Inglis has written a number of papers concerning VLSI chip designs with which he was involved. A few of the titles are "A8066 Write Pulse Programmable Timer", "Four Phase Clock Decoder", "Sensitivity of MBC/MBI Devices to Variations in Temperature and Supply Voltage", and "IDS-5000 Device Database Setup".

END

OF

TITLE